

高级语言程序设计

第 8 章

结构体和其它数据机制

华中师范大学物理学院 李安邦

本章概述

- 介绍 C/C++ 语言的其他数据描述机制，主要是结构体(struct)等。
- 各种数据机制的概念、意义和用途。
- 在写处理复杂数据的程序，在定义复杂数据类型和结构时，这些机制应用广泛。“数据结构”课程中将大量使用这些机制。
- 这里主要介绍概念，给出一些程序实例。并介绍一些相关的概念和技术。
- 后续课程和其他书籍中可以看到大量实例。

第8章 结构体和其它数据机制

8.1 定义类型

8.1.1 简单类型定义

8.1.2 定义数组类型

8.2 结构体 (struct)

8.3 结构体编程实例

8.4 链接结构体 (自引用结构体)

8.1 定义类型

语言为各种基本类型提供了**类型名**，可用于定义/说明变量，描述函数参数与返回值，类型强制等。

数组、指针等等可能使说明变得很复杂，使用不便。也不容易保证多个类型描述的一致性。

如果能把复杂类型描述看作类型（用户定义类型）加以命名，可带来很大方便，特别是在实现复杂程序/软件系统时。

定义类型是重要语言机制，定义好的类型最好能像内部类型一样使用。

C 语言类型定义机制较弱，主要作用是**简化描述**，增强程序可读性。C++ 功能更强。

8.1.1 简单类型定义

类型定义用关键字 `typedef`:

`typedef` 原有类型名 新类型名;

C/C++ 并不认为使用 `typedef` 定义的类型是新类型, 而认为它们只是原有类型的新名字 (别名)。

例: `typedef unsigned long double ULD;`

定义后的 ULD 可以像基本类型名一样用:

`ULD x, y, *p;`

`ULD fun1(double x, ULD y);`

这种类型定义可简化程序书写, 有一定实际价值。

定义新类型名还可以提高程序的可读性，使程序更清晰。

例：程序的返回值为整数时，它是表示算术计算结果，还是表示函数的工作状态？

例：`int func(int m, double x);`

可以把 `int` 定义一个别名 “Status”，然后用该别名说明函数的返回值类型：

```
typedef int Status;
```

```
Status func (int m, double x);
```

定义指针类型为新类型，可以提高程序的可读性，减少出错可能性。

例： `char * pa, pb;` `// pb 是什么类型？`

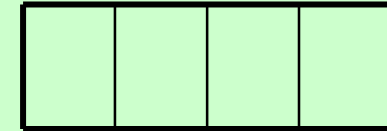
利用 `typedef` 把字符指针类型定义为新类型，可以减少出错可能性：

```
typedef char* pchar;  
pchar pa, pb;
```

8.1.2 定义数组类型

例如，定义一种具有4个元素的双精度数组类型：

```
typedef double Vect4[4];
```



此后可以写：

```
Vect4 v1, v2; // 定义两个数组变量
```

```
Vect4 *pvect; // 定义指针变量
```

```
double det(Vect4 v); // 用于说明函数参数
```


8.1.3 定义函数指针类型

函数指针参数名被有关描述的各种成分重重包裹，很不显眼，影响阅读和理解：

```
double chordroot(double (*pf)(double), double x1,  
double x2);
```

先用 typedef 定义好有关的函数指针类型，引进专门的类型名，再描述这种类型的函数指针参数或定义函数指针变量，都会变得很简单。

```
typedef double (* MathFun)(double);  
double chordroot(MathFun pf, double x1, double  
x2);
```

提醒：注意区分 `#define` 与 `typedef`
宏定义 类型定义

第8章 结构体和其它数据机制

8.1 定义类型

8.2 结构体 (struct)

8.2.1 结构体类型定义

8.2.2 结构体变量定义和初始化

8.2.3 结构体变量的使用

8.2.4 结构体与函数

8.2.5 结构体、数组与指针

8.3 结构体编程实例

8.4 链接结构体 (自引用结构体)

8.2 结构体 (struct)

被处理的数据常常形成一些逻辑整体，具有紧密的内在联系。

各成分的性质类似（同类型）时可以用数组。

数据体成分类型不同的情况很普遍。如居民身份证，成分有：姓名、性别、出生日期、住址……。这组信息共同描述了一个居民。无法用数组表示。

许多语言提供了将多个相同或不同类型的数据组合起来的机制，常称记录(record)，C/C++称为结构体(structure)。

结构体是复合数据对象，其中的数据项称为结构体的成分或成员。成员给定名字，通过成员名访问。

8.2.1 结构体类型

要定义一种结构体类型，就需要描述它的各个成员的情况，包括每个成员的类型及名字。



结构体类型描述由 struct 引导，基本形式：

struct 结构体标志 { 成员说明序列 };

形式与变量定义相同

结构体里可以有任意多个成员，成员可以是任何可用类型。

例如，平面上一个点的结构体：

```
struct Point {  
    double x, y;  
};
```

```
struct Point2 {  
    double crd[2];  
};
```

在 C 语言中，以上语句定义了一种**结构体类型**，在后续使用时要把“**struct**”**关键词和结构体标志写在一起**来使用这种结构体类型。

例如： **struct Point**

结构体中的成员也可以是已经定义好的结构体类型。
已定义的结构体类型可以用于定义其它结构体类型。

例：

```
struct Circle {  
    struct Point center;  
    double radius;  
};
```

为了使语句更简洁，通常用 typedef 把结构体类型重新命名为一个新的类型名和相应的指针类型名：

```
typedef struct Point Point;  
typedef struct Point * PtrPoint;
```

可以把结构体类型的定义和类型命名合写在一起：

```
typedef struct Point {  
    double x, y;  
} Point, *PtrPoint;
```

```
typedef struct Circle {  
    Point center;  
    double radius;  
} Circle, *PtrCircle;
```

《数据结构》课程里经常见到这种用法。

注意：

(1) 在定义结构体类型时，一个结构体类型中的成员名字不能重名，而不同的结构体类型中则完全可以包含名称相同的成员，彼此无关。

```
typedef struct Coord3d {  
    double x, y, z;  
} Coord, *PtrCoord;
```

(2) 在定义结构体类型时，结构体的成员不能是正在描述的这个结构体本身。

```
struct Invalid {  
    int n;  
    struct Invalid iv;    //错误  
};
```


附录 4 命名规范

本书中使用的命名规范如下：

- 1、常量（常变量、枚举常量和宏）使用全部大写的字母；
- 2、普通变量和函数参数通常用全小写字母。而且 i、j、k、m、n 等字母或以它们开头的变量名只用于说明整型变量（不用于说明实型变量），而 x、y 和 z 只用于说明实型变量。
- 3、函数名通常用全小写字母。当名称为“动词+名词”形式时使用小驼峰式命名（动词为小写，名词为首字母大写）。
- 4、结构体标志和用 `typedef` 定义的类型名称用大驼峰式命名（每个单词的首字母大写）。

8.2.2 结构体变量定义和初始化

在程序里使用结构体，那么就需要定义结构体变量（以结构体为类型的变量）。

可用“struct 结构体标志”，
或用 typedef 定义的类型名。

```
struct Point dot1, dot2;
```

```
Point dot3, dot4;
```

```
Circle cir1, cir2;
```

也可以定义相应的指针变量（可初始化）：

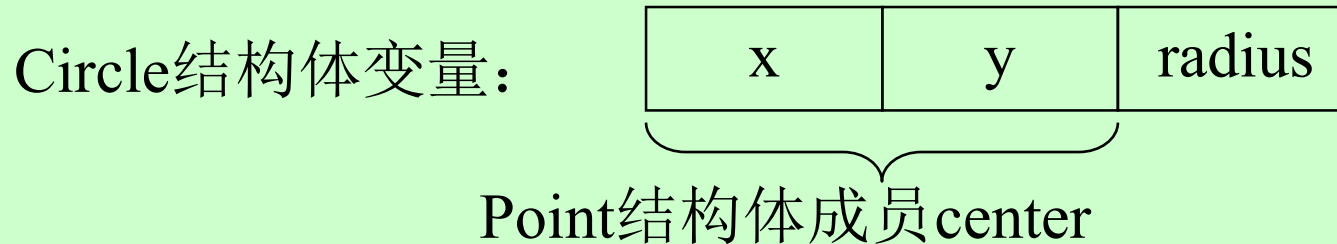
```
struct Point *pdot1;
```

```
PtrPoint pdot2 = &dot2;
```

所定义的结构体变量在内存中占用多大存储空间？

在计算机中存储一个结构体数据对象（例如结构体变量）就需要存储其中的各个成员。

编译系统将为每个结构体变量分配一块足够大的存储区，把它的各个成员顺序存于其中。



实际大小需要使用 `sizeof` 运算符计算出来才准确。

与简单类型变量和数组一样，结构体变量也可以在定义时直接初始化。形式与数组一样：

```
Point dot1 = {2.34, 3.28}, dot2;
```

```
Circle circ1 = {{3.5, 2.07}, 1.25}, circ2 = {12.35, 10.6, 2.56};
```

初始化描述中的各个值将顺序地提供给结构体变量的各个基本成员，初值表达式必须是可静态求值的表达式。

嵌套结构可以加括号。项数不得多于结构体变量所需，不够时剩下的成分自动初始化为 **0**。



未提供初始值时，**外部**和**静态局部**结构体变量的成员初始化为 **0**，自动变量不自动初始化。

初值表达式只能用于变量定义，不能出现在语句里。

8.2.3 结构体变量的使用

对结构体变量的操作：结构体成员访问和整体赋值

访问结构体成员的操作用圆点运算符 "." 描述。

具有最高的优先级，采用自左向右的结合方式。 

```
dot1.x = dot1.y = 0.0;    dot2.x = dot1.x + 2.4;
```

```
dot2.y = dot1.y + 4.8;    circ1.radius = 0.9;
```

当某个结构体的成员是另一个结构体时，就需要使用**多级**圆点运算符。

```
circ1.center.x = 2.0;  circ1.center.y = dot1.y + 3.5;
```

```
circ2.center.x += 2.8;    circ2.center.y += 0.24;
```

相当于访问一个具有相应类型的变量，操作由类型决定。

程序里也可以用 & 取得结构体变量或其成员的地址。

结构体变量可以**整体赋值**，显然，赋值时只能用同样类型的“结构体值”。

效果：**结构体中各个成员的分别赋值。**

```
circ2 = circ1;    dot2 = dot1;
```

```
circ1.center = dot1;
```

变量 circ2、dot2 各成员的值将分别与 circ1、dot1 对应成员的值完全一样。

C/C++ 语言规定不能对结构体做相等与不等比较，也不能做其它运算。

对于**结构体指针变量**，在引用其所指结构体变量的成员时，可以有两种方法。

(1) 先用 ***** 作对所指结构体变量进行**间接访问**，然后再用圆点运算符**引用其成员**。

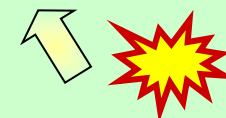
```
struct Point *pdot = &dot1; //定义指针变量并初始化
```

```
(*pdot).x = 0; (*pdot).y = 0;
```

由于圆点运算符的优先级高，此处必须写括号。

不写括号的描述 `*pdot.x` 和 `*pdot.y` 是错误的，它实际上表示的是 `*(pdot.x)` 和 `*(pdot.y)`。

(2) 为了方便从指针出发去访问结构体成员，C/C++ 语言为这种操作专门提供了运算符 “->” (“箭头运算符”) 。



pdot->x 相当于 (*pdot).x ,

pdot->y 相当于 (*pdot).y。于是可以写：

```
pdot->x = 0;
```

```
pdot->y = 0;
```

运算符 -> 也具有最高优先级（与圆点运算符、函数调用 [] 及数组元素访问 [] 一样），它也遵循从左向右的结合方式。

- 附录 1 C 和 C++ 语言运算符表

运算符	解释	结合方式
() [] -> .	括号（函数等），数组，两种结构体成员访问	由左向右
! ~ ++ -- + - * & (类型) sizeof	逻辑否定，按位否定，增量，减量，正负号， 间接访问，取地址，类型转换，求占用内存大小	由右向左
* / %	乘，除，取模	由左向右
+ -	加，减	由左向右

【例8-1】下面是一个展示结构体类型定义、结构体变量定义和使用的简单程序例子。它让用户输入平面上的一个点的坐标，输入一个圆的圆心和半径，然后判断该点是否在该圆的内部（即点与圆心之间的距离是否小于圆的半径）。

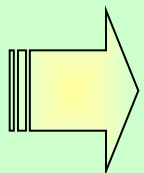
先定义坐标点结构体类型和圆结构体类型：

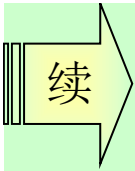
```
#include <iostream>
#include <cmath>
using namespace std;

typedef struct Point{
    double x, y;
} Point;
```

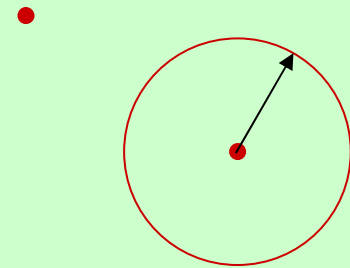
```
typedef struct Circle{
    Point center;
    double radius;
} Circle;
```

然后分别定义相应的变量，再进行输入和计算操作。





```
int main () {  
    Point dot1; Circle circ1 ; double dist;  
    cout << "Input a Point (x y): ";  
    cin >> dot1.x >> dot1.y;  
    cout << "Input a Circle (x y r): ";  
    cin >> circ1.center.x >> circ1.center.y >> circ1.radius;  
    dist = sqrt ((dot1.x-circ1.center.x) * (dot1.x-circ1.center.x)  
        + (dot1.y-circ1.center.y) * (dot1.y-circ1.center.y));  
    cout << "distance: " << dist << endl;  
    if (dist <= circ1.radius ) cout << "dot1 is inside circ1.\n";  
    else cout << "dot1 is NOT inside circ1.\n";  
    return 0;  
}
```



目 录

第8章 结构体和其它数据机制

8.1 定义类型

8.2 结构体 (struct)

8.2.1 结构体类型定义

8.2.2 结构体变量定义和初始化

8.2.3 结构体变量的使用

⇒ 8.2.4 结构体与函数

8.2.5 结构体、数组与指针

8.3 结构体编程实例

8.4 链接结构体 (自引用结构体)

8.2.4 结构体与函数

结构体类型既可以作为函数返回值类型，也可以作为函数的参数。

在书写上同样既可以用“**struct 结构体标志**”这种形式，也可以用 typedef 定义的类型名。

把结构体类型作为函数返回值类型时，只需要在函数声明或函数定义的返回值类型处写结构体类型。

例如：

```
struct Point mkPoint(double x, double y);
```

```
Circle mkCircle(double x, double y, double r);
```

使用函数处理存储在结构体中的数据时，既可以分散使用结构体成员，也可以整体使用结构体：

(1) 个别地将结构体成员的值传递给函数处理（可以用值参数、引用参数或指针参数等方式）。

(2) 当结构体参数的成员值不需要改变时，可以将整个结构体作为参数值传递给函数 → 结构体值参数

(3) 当结构体参数需要改变时，可以用 C++ 中的引用方式传递结构体参数 → 结构体引用参数

(4) 将结构体的地址传给函数，也就是说传递指向结构体的指针值。这称为结构体指针参数。

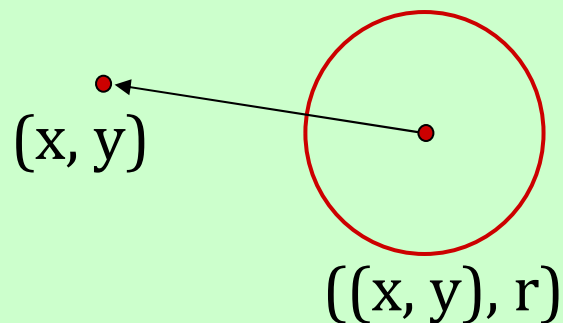
后三种方式都是把结构体作为整体来看待和处理，但正如针对其它参数的值传递和指针传递一样，这三种参数的作用方式和效果不同。

【例8-2】 请用户输入平面上的一个点的坐标，再输入一个圆的圆心和半径，然后判断该点是否在该圆的内部（即点与圆心之间的距离是否小于圆的半径）。要求写一系列的函数实现题目中的功能。

这个程序的功能有三部分：

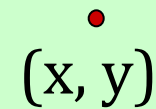
1. 给一个 Point 类型的变量赋值，
2. 给一个 Circle 类型的变量赋值，
3. 使用一个 Point 类型变量的成员和一个 Circle 类型变量的成员进行计算。

把三个功能分别写为三个函数，然后在主函数中进行调用。

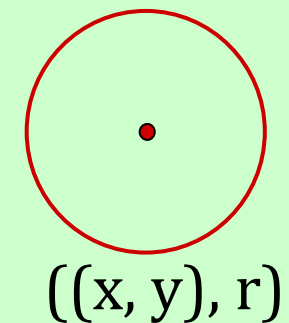


前两个功能中，是提供一些 double 参数值给结构体变量的成员赋值，并返回结构体变量：

```
struct Point mkPoint(double x, double y){  
    struct Point temp;  
    temp.x = x;  
    temp.y = y;  
    return temp;  
}
```



```
Circle mkCircle(double x, double y, double r) {  
    Circle temp;  
    temp.center.x = x;  
    temp.center.y = y;  
    temp.radius = r;  
    return temp;  
}
```



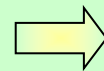
对于第三个功能“**计算平面上一个点与一个圆的距离**”，根据不同的参数形式，可以写出不同形式的函数。

(1) **把结构体的成员值作为参数**传递给函数：

```
double dist1 (double x1, double y1, double x2, double y2)
{ //值参数
    double dist;
    dist = sqrt ((x1-x2) * (x1-x2) + (y1-y2) * (y1-y2));
    return dist;
}
```

调用语句：

```
dist = dist1(dot1.x, dot1.y, circ1.center.x, circ1.center.y);
```



另外的方法是**把结构体作为一个整体作为参数**传递给函数。
可以写出如下三种写法：

```
double dist2 (Point dot, Circle circ) { //结构体值参数
    double dist;
    dist = sqrt ((dot.x - circ.center.x) * (dot.x - circ.center.x)
        + (dot.y - circ.center.y) * (dot.y - circ.center.y));
    return dist;
}
```

```
double dist3 (Point &dot, Circle &circ) { //结构体引用参数
    return sqrt ((dot.x - circ.center.x) * (dot.x - circ.center.x)
        + (dot.y - circ.center.y) * (dot.y - circ.center.y));
}
```

```
double dist4 (Point *dot, Circle *circ) { //结构体指针参数
    return sqrt ((dot->x - circ->center.x) * (dot->x - circ->center.x)
        + (dot->y - circ->center.y) * (dot->y - circ->center.y));
    //注意上式中 -> 与 . 的用法
}
```

主函数：

```
int main () {  
    Point dot1;  
    Circle circ1 ;  
    double x, y, r, dist;  
  
    cout << "Input a point (x y): ";  
    cin >> x >> y;  
    dot1 = mkPoint(x, y);  //!!!  
  
    cout << "Input the a circle (x y r): ";  
    cin >> x >> y >> r;  
    circ1 = mkCircle (x, y, r);  //!!!  
  
    dist = dist1(dot1.x, dot1.y, circ1.center.x, circ1.centery);  
    //成员参数  
    cout << "distance: " << dist << endl;
```

```
dist = dist2(dot1, circ1); //结构体值参数
cout << "distance: " << dist << endl;
dist = dist3(dot1, circ1); //结构体引用参数
cout << "distance: " << dist << endl;
dist = dist4(&dot1, &circ1); //结构体指针参数
cout << "distance: " << dist << endl;

if (dist <= circ1.radius )
    cout << "dot1 is inside of the circ1." <<endl;
else
    cout << "dot1 is NOT inside of circ1." <<endl;
return 0;
}
```

```
double dist1 (double x1, double y1, double x2, double y2);  
double dist2 (Point dot, Circle circ); //结构体值参数  
double dist3 (Point &dot, Circle &circ); //结构体引用参数  
double dist4 (Point *dot, Circle *circ); //结构体指针参数
```



→ 使用结构体成员作为参数时（**dist1**函数），由于结构体的每个成员都要写成一个参数，参数列表中的内容比较多。

→ 使用结构体值参数的方法（**dist2**函数）需要在运行函数时新建同类型的结构体变量、并进行变量值复制。在处理大型的结构体变量时，这种方法效率较低。

采用引用或指针的一个优点是可以避免复制整个结构体。建议读者在使用结构体整体作为函数参数时，**通常采用引用形式或指针形式为好。**

扩展说明：

C中的结构体只包含成员变量；

C++ 中的结构体与“类(Class)”更复杂一些，
可以包含成员函数。

成员变量和成员函数都是用圆点运算符 . 来调用。

`cin.get()`, `cin.clear()`, `cin.sync()`.

`infile.open()`, `infile.close`.

目 录

第8章 结构体和其它数据机制

8.1 定义类型

8.2 结构体 (struct)

8.2.1 结构体类型定义

8.2.2 结构体变量定义和初始化

8.2.3 结构体变量的使用

8.2.4 结构体与函数

⇒ 8.2.5 结构体、数组与指针

8.3 结构体编程实例

8.4 链接结构体 (自引用结构体)

8.2.5 结构体、数组与指针

结构体里可以包含数组成员。

例如学生信息结构体：

```
struct student {  
    int id;  
    char name[20];  
    char gender;  
    int birthyear;  
    int enteryear;  
    char department[20];  
    char major[20];  
}
```


另一方面，也可以定义以结构体作为元素的数组。

【例8-3】 在二维平面坐标上有 100 个 x 和 y 值都在 $[0, 100]$ 范围内的随机点，把它们的坐标依次全部输出到屏幕，并求它们的几何中心。

分别使用两种方法处理：

- (1) 采用两个数组分别保存 x 值和 y 值；
- (2) 使用平面点结构体数组。

方法（1）：定义两个长度为100的数组，分别保存 x 值和 y 值，然后再分别求平均值：

```

int main() {
    const int NUM = 100;
    double x[NUM], y[NUM], sumx = 0, sumy = 0;
    cout << "100 random points on surface. Using X Y arrays.\n\n"
    srand(time(0));
    for (int i = 0; i < NUM; i++) {
        x[i] = 1.0* rand() / RAND_MAX * 100;
        y[i] = 1.0* rand() / RAND_MAX * 100;
        cout << "i= "<<i << "\tx= " << x[i] << "\ty= " << y[i] <<endl;
    }
    for (int i = 0; i < NUM; i++ ) {
        sumx += x[i];      sumy += y[i];
    }
    cout <<"average x= "<< sumx / NUM << " average y=" << sumy / NUM;
    return 0;
}

```

方法（2）：使用表示二维平面点的**结构体数组**。

设想出程序的主体内容和基本流程如下：

定义结构体类型；

定义结构体数组；

使用随机数函数设定各点的坐标；

计算数组中的二维坐标平均值（几何中心）；

输出结果；

```
typedef struct Point {  
    double x, y;  
} Point;
```

定义结构体类型;

```
int main() {  
    const int NUM = 100;  
    Point pt[NUM], cent = {0,0};  
    srand(time(0));  
    for (int i = 0; i < NUM; i++) {  
        pt[i].x = 1.0* rand() / RAND_MAX * 100;  
        pt[i].y = 1.0* rand() / RAND_MAX * 100;  
        cout << i << ": (" << pt[i].x << ", " << pt[i].y << ")" << endl;  
    }
```

定义结构体数组;

使用随机数函数
设定各点的坐标;

```
for (int i = 0; i < NUM; i++ ) {  
    cent.x += pt[i].x;  
    cent.y += pt[i].y;  
}  
cent.x /= NUM;  
cent.y /= NUM;
```

计算数组中的二维坐标
平均值（几何中心）；

```
cout << "center : (" << cent.x << ", " << cent.y << ")" << endl;  
return 0;  
}
```

输出结果;

【例8-4】 假设有一个名为“atoms.txt”的数据文件，其中存储了一个大分子中各个原子的信息，数据在文件中分为四列，分别表示原子的质量和 XYZ 坐标，各列之间用制表符分隔，首行是文字注释，随后的每一行分别表示一个原子的信息，其中可能有空行。如下所示：

mass	x	y	z
1	58.839	78.090	61.915
12	58.510	77.806	61.006
1	59.119	78.220	60.245
12	58.674	76.279	60.852
16	58.167	75.595	61.738
14	59.467	75.715	59.849
...			

$$x_c = \frac{1}{M} \sum_{i=1}^n m_i x_i$$

$$y_c = \frac{1}{M} \sum_{i=1}^n m_i y_i$$

$$z_c = \frac{1}{M} \sum_{i=1}^n m_i z_i$$

整个文件的行数（原子个数）事先未知。请编写程序读取该文件中的数据到一个原子结构体数组中，然后计算该分子的质量中心。

此题将使用结构体数组，还复习：

- 文件读取
- 动态分配和释放内存

设想出程序的主体内容和基本流程如下：

定义原子结构体类型；

打开文件读取一次，获得原子个数 n ，关闭文件；

定义原子结构体指针，动态分配数组内存；

重新打开文件，读取原子信息，关闭文件；

计算分子的质量中心；

销毁动态分配的数组；

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

typedef struct Atom{
    double m, x, y, z;
} Atom;

int main() {
    const int SIZE = 100;
    char str[SIZE];    int n = 0;  //原子个数
    char fname[] = "atoms.txt";  //文件名存储于字符数组中
    ifstream in;
    in.open(fname);  //打开文件以供读取
    cout << "读取数据文件: " << fname << endl;
    in.getline(str, SIZE);  //读入文件首行注释
```



```

while (in.getline(str, SIZE)) { //逐行读入进行计数
    if (strlen(str) !=0)    n++;
    //cout << n << ": " << str << endl;
}
in.close(); //关闭文件
cout << "原子总数: " << n << endl;

```

```

in.open(fname); //重新打开文件再次读取
in.getline(str, SIZE); //读入文件首行注释
Atom *at = new Atom[n]; //定义指针变量并申请分配空间
//Atom at[n]; //定义以变量为长度的数组 (C99或C++)
Atom cent = {0, 0, 0, 0}; //质心
for (int i = 0; i < n; i++) { //读取原子信息
    in >> at[i].m >> at[i].x >> at[i].y >> at[i].z; //数组写法
    //in >> (at+i)->m >> (at+i)->x >> (at+i)->y >> (at+i)->z; //指针写法
    //cout << i << ": " << at[i].m << " " << at[i].x << " "
    //    << at[i].y << " " << at[i].z << endl;
}
in.close(); //关闭文件

```

```
for (int i = 0; i < n; i++) { //计算质心
    cent.m += at[i].m;
    cent.x += at[i].m * at[i].x;
    cent.y += at[i].m * at[i].y;
    cent.z += at[i].m * at[i].z;
}
cent.x /= cent.m;
cent.y /= cent.m;
cent.z /= cent.m;

cout << "质心坐标: (" << cent.x << ", "
    << cent.y << ", " << cent.z << ")\n";
delete []at; //释放动态分配的数组存储空间

return 0;
}
```

目 录

第8章 结构体和其它数据机制

8.1 定义类型

8.2 结构体 (struct)

8.3 结构体编程实例

8.3.1 复数的表示和处理

从底向上开发

8.3.2 学生成绩管理系统

从顶向下开发

8.4 链接结构体 (自引用结构体)

8.3.1 复数的表示和处理

C 语言提供了许多数值类型，但也不完全，例如缺少复数类型。要写处理复数数据的程序时该怎么办？当然可以用两个 **double** 表示一个复数，定义函数：

```
addcomplex(double r1, double i1, double r2, double i2);
```

结果很难返回。改为：

```
addcomplex(double r1, double i1, double r2, double i2  
            double *rr, double *ri);
```

参数多，需要记住各个位置，使用很麻烦。

注意，这里的一个复数是一个逻辑数据体，应定义为类型，再定义一批以复数类型为操作对象的函数。

定义为类型后，程序其他部分就会变得清晰简单。

人们在程序设计实践中认识到，设计实现一个较复杂的程序时，最重要的是找出所需的一批数据类型。将它们的结构和功能分析清楚，设计并实现。在这些类型的基础上实现整个程序。

这样得到的程序更清晰，各个部分功能划分较明确，更容易理解和修改。

考虑复数类型的实现。

复数可有多种数学表示方式：平面坐标，极坐标等。

某种表示可能更适合某个特定应用，需仔细斟酌。

作为例子，这里选择平面坐标，实部和虚部。考虑复数运算，让复数具有两个**double**成分。

应该用什么机制将这两部分结合起来呢？

两部分类型相同，可以用两个**double**元素的数组，或用两个**double**成员的结构。

由于需要定义许多运算，用结构表示有利于将复数作为参数传递和作为结果返回。因此做下面定义：

```
typedef struct Complex{  
    double re, im;  
} Complex;
```

下面考虑基于这个类型的运算。

由于**Complex**对象的数据项很少，可以考虑直接传递**Complex**类型的值和结果，避免复杂存储管理问题。

基本的算术函数，原型：

```
Complex addCx(Complex x, Complex y);  
Complex subCx(Complex x, Complex y);  
Complex tmsCx(Complex x, Complex y);  
Complex divCx(Complex x, Complex y);
```

还需考虑如何构造复数。我们不希望使用复数的程序直接访问 **Complex** 的成分，否则程序里的错误将很难控制。如果所有使用都经过我们定义的函数，只要这些函数正确，程序里的正确使用就有保证了。

下面是几个构造函数：

```
Complex mkCx(double re, double im);  
Complex d2Cx(double d);  
Complex n2Cx(int n);
```

后两个函数也可看作由 **double** 和 **int** 到复数的“数值转换”函数，定义它们是为了使用方便：

```
Complex mkCx(double r, double i) {  
    Complex c;  
    c.re = r; c.im = i;  
    return c;  
}
```

```
Complex d2Cx(double d) {  
    Complex c;  
    c.re = d; c.im = 0;  
    return c;  
}
```

```
Complex n2Cx(int n) {  
    Complex c;  
    c.re = n; c.im = 0;  
    return c;  
}
```


加法函数的定义：

```
Complex addCx(Complex x, Complex y) {  
    Complex c;  
    c.re = x.re + y.re; c.im = x.im + y.im;  
    return c;  
}
```

减法函数与此类似。乘法函数的算法复杂一点，根据数学定义也不难给出。

除法函数有个新问题：除数为**0**时该怎么办？

复数除法的数学定义是（***c*** 和 ***d*** 都为 **0** 时出问题）：

$$\frac{a + bi}{c + di} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

C 语言内部类型除 **0** 的规定是“其行为没有定义”，编程者自己负责。

我们实现复数操作时有两种选择：

- 沿用 **C** 方式，要求用复数类的人保证不出现除 **0**
- 检查除 **0** 的情况，提供动态信息并返回某个特殊值

采用第一种方式时可直接按公式定义函数。

下面函数定义检查除 **0** 情况，输出错误信息并返回 **1** 的复数。

```
Complex divCx(Complex x, Complex y) {  
    Complex c;  
    double den = y.re * y.re + y.im * y.im;  
  
    if (den == 0.0) {  
        fprintf(stderr, "ComplexErr: div 0.\n");  
        c.re = 1; c.im = 0;  
    }  
    else {  
        c.re = (x.re*y.re + x.im*y.im) / den;  
        c.im = (x.im*y.re - x.re*y.im) / den;  
    }  
    return c;  
}
```

为了方便，还可以定义复数的输出和函数。输出函数向某个流输出一个复数。实现这个函数前先要为复数确定一种输出形式，函数定义：

```
void prtCx(FILE *fp, Complex x) {  
    fprintf(fp, "(%f, %f)", x.re, x.im);  
}
```

输入函数：

```
int readCx(const char *prompt, Complex *pcx);
```

pcx 的实参应是**Complex**地址。函数设计应参考标准库输入函数：实际完成复数输入时返回**1**，转换失败时返回**0**，遇文件结束出错误时返回 **EOF**。**IO** 函数最好统一设计，使 **prtCx** 输出能由 **readCx** 输入。

写一个 main 函数测试这些函数 ……

8.3.2 学生成绩管理系统

【例8-6】 现在考虑用结构体重新实现前面的学生成绩实例（参见“6.6.1 学生成绩统计分析”）。

除了作为结构体的编程实例外，我们还想通过这一实例介绍实践编程中的更多概念和常见用法。

原来的学生成绩记录文件仅包含学生的一门课程的一次考试的成绩，这个格式并不具有实用性（因为它没有记录学生信息，也没有记录平时成绩、期末考试成绩和总评成绩）。文件里应该同时还包含每个学生的信息（常用的是学号、姓名、性别和出生年份）才行。

为此需要在程序中定义一种结构体，其中包含学生的信息（常用的是学号、姓名、性别和出生年份）和课程成绩（为了减轻任务的难度，我们这里只考虑一门课程的成绩，它包括三部分：平时成绩、期末考试成绩和总评成绩）。在此对每一项数据都要进行事先分析。

- 学生的**学号**：在此选用把学号定义为整数。
- 学生的**姓名**肯定是字符串，但是长度不一，而且可能有空格：为了简化任务，选用定长字符数组存储。长度设为20个字符。在读写时要注意姓名中的空格。
- **性别**：选用字符。
- **出生年份**，很显然可用整数表示。
- **平时成绩、期末考试成绩和总评成绩**用实数表示。选用float类型即可。

```
typedef struct StuRec{  
    int no; //学号  
    char name[20]; //姓名  
    char gender; //"性别"的英文单词为gender  
    int birthyear; //出生年份  
    float score1, score2, score3; //平时成绩、期末考试成绩和总评成绩  
} StuRec;
```

- 程序中是为了处理一批学生的信息和成绩，可以在程序中定义一个较大的学生记录结构体数组或采用动态分配的结构体数组。在此为了简化任务，定义一个结构体数组吧。应该定义一个表示学生最大数量的常数MAXNUM；我们采用下面定义：

```
enum { MAXNUM = 400 } //最大学生数量
```

```
StuRec students[MAXNUM]; //全局性的学生记录结构体数组
```

实际的学生数量肯定是小于等于MAXNUM。我们用一个全局变量num表示实际学生数量：

```
int num = 0; //实际学生数量（初始化为0）
```

- 程序有必要记录全体学生的学校、院系和班级信息，所以定义一个全局变量title：

```
char title[256]="null";//全体学生的学校、院系和班级信息（故意初始化为某个名称）
```

- 学生成绩管理系统的主要功能是可以让用户输入学生的信息和成绩，并存储在文件中，文件名应该在首次录入时由用户提供，所以定义一个全局的数据文件名（并初始化为某个特殊名称）：

```
char datafile[256] = "null"; //存储学生信息和成绩的数据文件名
```

教师在输入学生成绩时，只需要输入平时成绩和期末考试成绩，这两项成绩按照事先确定的所占比重（例如，前者占40%，后者占60%）计算出总评成绩，而且按照一定的及格分数线分别统计。

定义一个表示平时成绩所占比重全局变量rate1（期末考试成绩所占比重为 1-rate1，不必额外定义）和一个表示及格分数级的全局变量passline:

```
double rate1 = 0.4; //平时成绩比重。期末考试成绩比重为 1-rate1
double passline = 60; //及格分数线
这两个数据应该是允许用户修改的。
```

为了绘制统计直方图，定义相关的其它常量。

```
enum {
    HISTOHIGH = 60, //直方图最大高度
    SEGLEN = 5,     //直方图间隔宽度
    HISTONUM = (100/SEGLEN)+1 //直方图间隔数
};
```


有了上面的基本设计，下面就可以考虑程序的整体功能了。

学生成绩管理系统的主要功能可以让用户输入学生的信息和成绩，输入时应该可以批量逐个输入或者个别修改。程序每次启动时都应该能够读入原有文件中的数据。除此之外，还应该允许用户修改一些参数（成绩比例和及格分数），这些参数事先存储在一个参数配置文件中（例如这个文件取名为“config.ini”）。

通常来说，学生的信息管理和成绩管理是在两个不同时段进行的：在课程授课阶段录入学生信息（这时把学生成绩都赋为 -1，以指示并未录入成绩），在期末考试之后录入学生成绩。而且在录入学生信息或成绩时可能是批量进行的，或单独增加和修改的。因此，可以设想，程序的主要流程如下：

从参数配置文件中读取参数；

显示如下菜单：

- 1.批量输入学生信息
- 2.单独增加/修改学生信息
- 3.批量输入学生成绩
- 4.单独增加/修改学生成绩
- 5.列出所有学生信息和成绩
- 6.成绩统计分析
- 7.修改参数
- 0.退出

用户输入菜单项数字，执行相应的功能；

保存各项参数到配置文件并退出；

大致地写出主函数的内容。

```
int main() {
    int choose = -1;
    ReadConfig(); //从参数配置文件中读取程序参数
    if (strcmp(datafile, "null")) //数据文件名不等于初始化的空值时就读取数据
        ReadData();
    else
        cout << "没有学生数据。" << endl;
    while(choose != 0) {
        cin.sync();
        cin.clear();
        cout << "==== 学生成绩管理系统 =====" << endl;
        cout << "1.批量输入学生信息      " << endl;
        cout << "2.单独增加/修改学生信息  " << endl;
        cout << "3.批量输入学生成绩      " << endl;
        cout << "4.单独增加/修改学生成绩  " << endl;
        cout << "5.列出所有学生信息和成绩 " << endl;
        cout << "6.成绩统计分析          " << endl;
        cout << "7.修改参数              " << endl;
        cout << "0.退出                  " << endl;
        cout << "===== " << endl;
    }
```

```

do {
    cout<< "请选择程序功能(1-7, 0): ";
    cin >> choose;
} while(choose <0 || choose >7);
cin.sync();
cin.clear();
switch(choose) {
    case 1: { InputStud(); break; } //1.批量输入学生的信息
    case 2: { ModifyStud(); break; } //2.单独增加/修改学生信息
    case 3: { InputScore(); break; } //3.批量输入学生成绩
    case 4: { ModifyScore(); break; } //4.单独增加/修改学生成绩
    case 5: { ShowData(); break; } //5.列出所有学生信息和成绩
    case 6: { Statistic(); break; } //6.成绩统计分析
    case 7: { SetConfig(); break; } //7.修改参数
    default: break;
}
}
SaveConfig(); //把程序参数保存到参数配置文件
return 0;
}

```

- 分析主程序的功能，我们首先注意到，“从参数配置文件中读取程序参数”和“把程序参数保存到参数配置文件”这两个功能是完全相关的，读取时必须严格按照保存时的顺序和格式来进行读取。而且读取之后有必要在屏幕上显示所有参数，于是就额外编写一个用于显示参数的函数。

`void ShowConfig()`

`void ReadConfig()`

`void SaveConfig()`

- 菜单中的第6项功能“修改参数”也与此相关，

`void SetConfig()`

在开始编写录入学生信息和录入学生成绩的函数之前，首先要考虑数据文件读写。在从文件读写数据时，要特别注意数据项的分隔和含有空格的字符串的读写。如果数据项彼此用空格分隔，则在读取时会难以正确地读取含有空格的学生姓名。合理的解决办法是让数据项彼此用制表符分隔。而且读数据的函数应该与写数据的函数完全匹配才行。我们写出从文件读写学生信息和成绩的函数如下：

```
void SaveData()
```

```
void ReadData()
```

莫斯科大学物理系 2018 级

4

20181001	张三	F	2001	-1	-1	-1
20181002	李四	F	2002	-1	-1	-1
20181004	Andrew B Rubin	M	2002	-1	-1	-1
20181003	Alexei V Finkel	M	2003	-1	-1	-1

接下来就可以分别编写录入学生信息和学生成绩的函数了。这两个函数的主要功能就是依次接收用户所输入的各项数据。接受用户输入是一个常规工作，容易编写，只是其中需要注意能够接收学生姓名中的空格。还要注意的是，为了让用户正确地输入数据、减少出错的可性能，有必要在屏幕上显示必要的提示信息。于是我们写出这两个函数如下：

```
void InputStud()
```

```
void InputScore()
```

此例的细节比较多，课堂上就不详细介绍了。

目 录

第8章 结构体和其它数据机制

8.1 定义类型

8.2 结构体 (struct)

8.3 结构体编程实例

8.4 链接结构体 (自引用结构体)

8.4.1 链接结构体

8.4.2 自引用结构体的定义

8.4.3 程序实现

8.4.4 数据与查找

9.5.1 链接结构

问题的提出（词频统计）

设要统计正文文件里各个单词出现的次数。

典型应用：语言学要统计单词出现频率，分析文献或计算机程序等都要做类似统计工作。

新情况：统计前不知道有多少不同的词，无法在编程时准备好统计中使用的完整数据结构。

可能方案：动态分配计数器数组，必要时调整大小（用**realloc**）。问题：新词逐个遇到，反复调整分配效率比较，而且需要使用很大的块，不够灵活。

如果词很多，能否找到足够大的存储块也是问题。

希望有一种能方便地动态变化的组织结构，满足被存储数据项动态增加/减少的需要。链接结构。

链接结构通过指针、结构（**struct**）和动态存储管理实现，其基本构件是自引用结构。自引用结构的对象分为两部分：

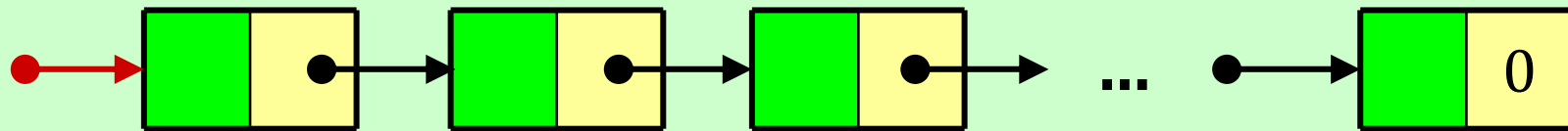
结构中的各种实际数据成员

一个或几个指向本类结构的指针

一个结构通过指针引用同类结构，多个结构通过指针建立联系。指向结构的指针称为链接，形成的复杂数据结构称为链接结构。

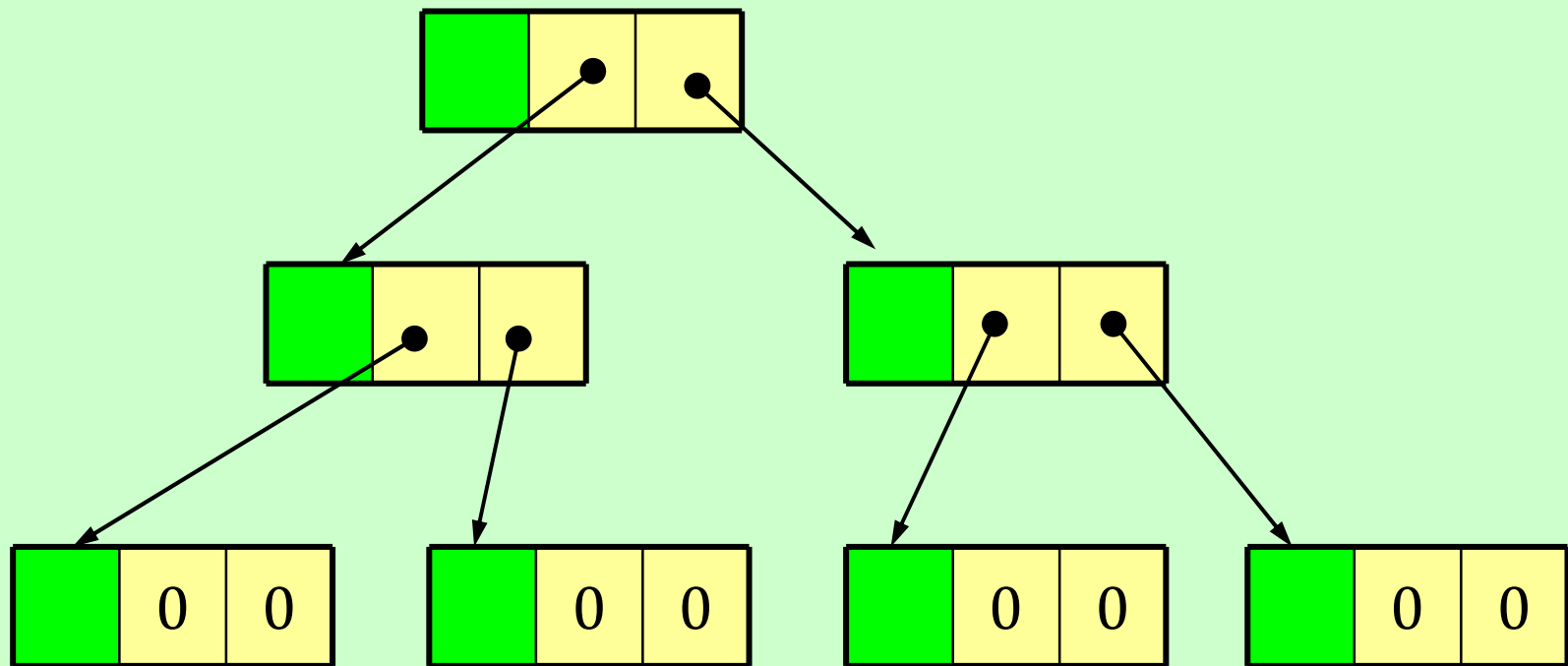
最简单的链接结构是线性链接形成的表：**链接表**。

每个自引用结构有一个**链接指针**，一批结构。一个链接到一个形成序列：



链接表就像链条，自引用结构是链节，表结点，结点间由指针连接形成整个结构。所有结点（结构）由动态分配创建。从指向表首结点的指针出发，沿链接可顺序访问表中各结点。该指针代表整个表。通常把最后结点的指针置空表示结束。

以自引用结构为基本构件可构造出许多复杂的数据结构。另一典型结构是二叉树，其中的每个结点有两个链接指针。下面是一个二叉树结构。《数据结构》课将进一步讨论这方面问题。



9.5.2 自引用结构的定义

以词频统计为例。用链接表作为基本数据结构。

对应于词的相关信息包括词本身及其计数。为简单，假设词长不超过**19**个字符（若无此限，就要设法保存和处理任意长的词。这个问题可作为一个大程序练习），

设整数表示范围足以应付词的统计（否则考虑用其他类型，如 **unsigned long**）。

在这些假设下，在统计用的结构里，两个基本数据成员可定义为：

```
char word[20];  
int count;
```

自引用结构里有一个指向本类结构的指针。

应该把所需自引用结构定义为类型。不但要定义结构类型，还应定义一个指向结构的指针类型。

```
typedef struct Node{  
    char word[20];  
    int count;  
    struct node *next;  
} Node, *LinkListList;
```



有了结构类型定义后，主函数部分：

```
#include <stdio.h>
#include <ctype.h> /* 需要判断字符类型 */
#include <string.h> /* 要做串复制和比较 */
#include <stdlib.h> /* 做动态存储分配 */
enum { MAXLEN = 20 };

typedef struct Node {
    char word[MAXLEN]; int count;
    struct Node *next;
} Node, *LinkListList;

int getword(int limit, char w[]);
LinkListList addword(LinkListList l, char w[]);
void printwords(LinkListList l);

LinkListList list = NULL; /* 表的头指针 */
char word[MAXLEN]; /* 临时字符数组 */
```

8.4.3 程序实现

设计好了链表结构体之后，开始考虑程序的主函数的功能。显然，根据题目要求，可以设想出主函数的主体内容和基本流程如下：

- 打开文件;

- 循环地从文件中读取单词;

- 把单词添加到链表中;

- 关闭文件;

- 打印链表;

在这几部分功能中，“打开文件”和“循环地从文件中读取单词”的功能可以参考前文第120页“4.4.3 编程实例3：文件中的单词计数”和第210页“6.6.2 统计C源程序中的关键字”。

程序中的主体结构函数部分可以写为：

```
#include <iostream>
#include <cstdlib> //动态存储分配
#include <cstring> //字符相关函数
#include <fstream> //文件输入输出流
using namespace std;
enum { MAXLEN = 20 };
// 结构体类型定义
typedef struct Node{
    char word[MAXLEN];
    int count;
    struct node * next;
} Node, *LinkListList;
// 有关函数的原型说明
int getword(char w[], int limit); //从文件中读取得到一个单词
LinkListList addword(LinkListList l, char w[]); //向链表增加单词（新
    结点或旧结点计数加1）
void printwords(LinkListList l); //打印链表中所有结点中的单词及其
    计数
ifstream inFile; // 全局的文件读入流
```



```

int main () {
    LinkListList list = NULL; // 链表的头指针
    char word[MAXLEN]; // 读入用的临时字符数组

    char filename[56]="plain.txt";
    inFile.open (filename); //打开文件输入流
    if (!inFile) { //如果打开文件失败，则 inFile 得到一个空指针
        cout << "错误：无法打开数据文件 " << filename << "。程序异常退出。 \n";
        exit(1); // 打开文件失败，则显示错误信息并退出程序。
    }
    while (getword(word, MAXLEN) != 0)
        list = addword(list, word);
    inFile.close(); //关闭文件输入流
    printwords(list);
    return 0;    addword完成对一个词的统计动作，返回修改后的统计表。
}
                主函数对每个词调用addword，不断更新统计表。

```

在几个函数里，getword不是新东西，这里也要求它把读入的单词存入参数数组里，最后返回单词的长度，长度为0表示再也没有新单词了，程序的工作可以结束。在getword返回时，数组word里的有效字符一定不超过19个，函数还应在有效字符后面放一个空字符'\0'。读者可以参考前文“6.6.2 统计C源程序中的关键字”中的getident函数。

printwords用循环实现，借助一个指针（参数也是局部变量）实现对表各结点的顺序访问：

```
void printwords(LinkListList p) {  
    for ( ; p != NULL; p = p->next)  
        printf("%d %s\n", p->count,  
                p->word);  
}
```

表最后空指针用于循环终止判断。

链接表基本处理方式：用一个指针从表头结点开始顺序处理结点，利用链接指针，直到表结束。

这种指针称为扫描指针。参数**p**就是扫描指针，函数调用时它得到表头结点地址。

LinkListList addword(LinkListList l, char w[]);

addword是最关键的部分，第一个参数是统计表（或部分），第二个参数是当时处理的词。

addword完成一个词的统计，返回修改过的表。主函数对每个词调用**addword**，完成统计表更新。

若一个词是首次遇到，就为它建新结点，记录词本身和统计值**1**；不是新词将结点统计值加一。

下面用递归方式定义**addword**。处理中如果可能修改表，用递归做就特别方便。

也可以用循环解决，程序复杂一些（见后）。

为简洁，定义辅助函数 **mknnode** 建立结点：它申请存储块，并把结点有关信息存进去：

```
LinkListList mknnode(char w[]) {  
    LinkListList p = (LinkList)malloc(sizeof(NODE));  
    if (p != NULL) {  
        strncpy(p->word, w, MAXLEN);  
        p->count = 1;  
        p->next = NULL;  
    }  
    return p;  
}
```

注意：1) 检查分配成功与否；2) 把新分配结点的链接指针域置空，使之处于确定状态。

函数 **addword** 已容易写了。定义：

```
LinkListList addword(LinkListList p, char w[]) {  
    if (p != NULL) {  
        if(strcmp(p->word, w) == 0)  
            p->count++;  
        else  
            p->next = addword(p->next, w);  
        return p;  
    }  
    else  
        return mknode(w);  
}
```

用循环方式重新写函数**addword**

现在不再返回修改后的表，而是直接修改原表。这个函数总是从全局变量**list**开始。

函数原型可以改为：

```
void addword(char w[]);
```

对**addword**的调用形式也要改变。

这里也需要一个扫描指针。

特殊问题是（初始时）空表的处理。一般情况下新建结点总连在所有结点之后（更改最后结点的指针）。初始时第一个结点应连在表头指针上。

```
void addword(char w[]) {  
    LinkListList p = list;  
    if (p == NULL) { /* 整个统计表为空 */  
        list = mknnode(w); return;  
    }  
    while (1) {  
        if (strcmp(p->word, w) == 0) {  
            p->count++; break; /* 已有词 */  
        }  
        if (p->next == NULL) { /* 新词 */  
            p->next = mknnode(w); break;  
        }  
        p = p->next; /* 继续扫描 */  
    }  
}
```


用不同 **getword** 可完成不同统计工作。如：

- 取空格分隔的单词，可以用于文字材料的词频统计；
- 取下一标识符，可用于统计程序中标识符使用情况；
- 取得下一个数，取得下一个正数，等等。

由于限制了词的最大字符数，对特别长的词这个程序肯定无法正确统计。这里未给 **getword**，无法分析会出什么错误。请结合自己实现的 **getword** 函数做分析。

存储空间限制可能引起统计错误。请考虑：

- 如果程序运行中存储申请失败会出什么问题？
- 会不会出现严重的程序运行错误？
- 会不会导致非法指针访问？
- 统计结果会出什么错？
- 如何在出问题时提供信息，在哪里修改？

8.4.4 数据与查找

本程序的一个缺点是工作效率。随着新词增加计数器表不断增长，表长等于不同单词个数。

文件小统计表不长。若不同词很多，效率会突出出来。

设一个文件里包含 **1000** 万词，**1** 万个不同词。统计表最后为 **1** 万个结点。设表均匀增长，平均 **5000** 结点。处理一个词平均查半个表，**2500** 次字符串比较。完成这一文件的处理工作要做 **250** 亿次比较。

若所用计算机能做每秒 **100** 万次字符串比较，整个工作需要 **25000** 秒，约 **7** 小时。

1000 万个词并不大。要处理规模更大的问题，对处理方法要进一步研究。

存储信息和查找是计算机应用中的典型问题，人们对它进行了许多研究，提出了各种提高效率的方法。练习中提出了一些改进的方法。另一个常见做法是采用前面简单介绍过的树型结构。

人们还提出了许多处理这种问题的数据表示方式和计算方式。后续《数据结构》课有进一步讨论。